QuickBean Documentation

Release 1.5.2

Olivier Kozak

February 01, 2015

Contents

QuickBean is a library that reduces the boilerplate code required to define beans.

CHAPTER 1

Installation

Here is how to install QuickBean :

```
pip install quickbean
```

CHAPTER 2

Starting with QuickBean

Suppose you have defined the following bean :

```
>>> class MyObject(object):
>>>     def __init__(self, my_property, my_other_property):
>>>         self.my_property = my_property
>>>         self.my_other_property = my_other_property
```

If you would like your bean to have a human-readable representation, you have to override the *__repr__* method :

```
>>> class MyObject(object):
>>>     def __init__(self, my_property, my_other_property):
>>>         self.my_property = my_property
>>>         self.my_other_property = my_other_property
>>>
>>>     def __repr__(self):
>>>         return 'MyObject(my_property=%s, my_other_property=%s)' % (self.my_property, self.my_othe
```

If you would like your bean to be equality comparable, you also have to override the *__eq__* and *__ne__* methods :

```
>>> class MyObject(object):
>>>     def __init__(self, my_property, my_other_property):
>>>         self.my_property = my_property
>>>         self.my_other_property = my_other_property
>>>
>>>     def __repr__(self):
>>>         return 'MyObject(my_property=%s, my_other_property=%s)' % (self.my_property, self.my_othe
>>>
>>>     def __eq__(self, other):
>>>         return other.__class__ is MyObject and other.__dict__ == self.__dict__
>>>
>>>     def __ne__(self, other):
>>>         return not self.__eq__(other)
```

Although there is nothing difficult here, it would be better if this boilerplate code could be automatically generated for you. This is exactly what QuickBean brings to you :

```
>>> import quickbean
>>>
>>> @quickbean.AutoBean
>>> class MyObject(object):
>>>     def __init__(self, my_property, my_other_property):
>>>         self.my_property = my_property
>>>         self.my_other_property = my_other_property
```

You may even let QuickBean generate the *__init__* method for you :

```
>>> import quickbean
>>>
>>> @quickbean.AutoInit('my_property', 'my_other_property')
>>> @quickbean.AutoBean
>>> class MyObject(object):
>>>     pass
```

CHAPTER 3

Contents

3.1 API reference

class quickbean.**AutoInit**(**properties*)

A decorator used to enhance the given class with an auto-generated initializer.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoInit('property_', 'other_property')
... class TestObject(object):
...     pass
```

You will get an auto-generated initializer taking all the declared properties :

```
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
... )
>>>
>>> test_object.property_
'value'
>>> test_object.other_property
'otherValue'
```

Note that it is also possible to declare default values :

```
>>> @quickbean.AutoInit('property_', ('other_property', 'defaultValue'))
... class TestObject(object):
...     pass
>>>
>>> test_object = TestObject(
...     property_='value',
... )
>>>
>>> test_object.property_
'value'
>>> test_object.other_property
'defaultValue'
```

Or, if you prefer something more explicit :

```
>>> @quickbean.AutoInit('property_', quickbean.Argument('other_property', default='defaultValue'
... class TestObject(object):
...     pass
>>>
>>> test_object = TestObject(
...     property_='value',
... )
>>>
>>> test_object.property_
'value'
>>> test_object.other_property
'defaultValue'
```

class quickbean.**AutoInitFromJson**

A decorator used to enhance the given class with an auto-generated JSON decoder.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoInitFromJson
... class TestObject(object):
...     def __init__(self, property_, other_property):
...         self.property_ = property_
...         self.other_property = other_property
```

You will get an auto-generated JSON decoder :

```
>>> test_object = TestObject.from_json_str('{"other_property": "otherValue", "property_": "value
>>>
>>> str(test_object.property_)
'value'
>>> str(test_object.other_property)
'otherValue'
```

Many frameworks handle JSON with dictionaries, so you never have to parse JSON strings directly. In that case, you should use the 'from_json_dict' method instead :

```
>>> test_object = TestObject.from_json_dict({'other_property': 'otherValue', 'property_': 'value
>>>
>>> str(test_object.property_)
'value'
>>> str(test_object.other_property)
'otherValue'
```

This decorator relies on the standard JSON decoder (https://docs.python.org/2/library/json.html). Values are then decoded according to this JSON decoder. But sometimes, it may be useful to customize how to decode some particular properties. This is done with types. Types are entities set to fields named as the corresponding properties suffixed with '_json_type' that decode the taken property value through the available 'from_json_str' method :

```
>>> class CustomJsonType(object):
...     # noinspection PyMethodMayBeStatic
...     def from_json_str(self, value):
...         return '%sFromJson' % json.loads(value)
>>>
>>> @quickbean.AutoInitFromJson
... class TestObject(object):
...     def __init__(self, property_, other_property):
...         self.property_ = property_
```

```
...         self.other_property = other_property
...
...     other_property_json_type = CustomJsonType()
>>>
>>> test_object = TestObject.from_json_str('{"other_property": "otherValue", "property_": "value
>>>
>>> str(test_object.property_)
'value'
>>> str(test_object.other_property)
'otherValueFromJson'
```

If you prefer handle JSON with dictionaries, use the 'from_json_dict' method instead :

```
>>> class CustomJsonType(object):
...     # noinspection PyMethodMayBeStatic
...     def from_json_dict(self, value):
...         return '%sFromJson' % value
>>>
>>> @quickbean.AutoInitFromJson
... class TestObject(object):
...     def __init__(self, property_, other_property):
...         self.property_ = property_
...         self.other_property = other_property
...
...     other_property_json_type = CustomJsonType()
>>>
>>> test_object = TestObject.from_json_str('{"other_property": "otherValue", "property_": "value
>>>
>>> str(test_object.property_)
'value'
>>> str(test_object.other_property)
'otherValueFromJson'
```

And if you have inner objects to map, you may simply use types like this :

```
>>> @quickbean.AutoInitFromJson
... class InnerTestObject(object):
...     def __init__(self, property_, other_property):
...         self.property_ = property_
...         self.other_property = other_property
>>>
>>> @quickbean.AutoInitFromJson
... class TestObject(object):
...     def __init__(self, inner):
...         self.inner = inner
...
...     inner_json_type = InnerTestObject
>>>
>>> test_object = TestObject.from_json_str('{"inner": {"other_property": "otherValue", "property
>>>
>>> str(test_object.inner.property_)
'value'
>>> str(test_object.inner.other_property)
'otherValue'
```

Sometimes, it may be very useful to directly decode a list of objects instead of having to decode them one by one. This is done through the 'list_from_json_str' method :

```
>>> @quickbean.AutoInitFromJson
... class TestObject(object):
...     def __init__(self, property_):
...         self.property_ = property_
>>>
>>> test_objects = TestObject.list_from_json_str('[{"property_": "value"}, {"property_": "otherV
>>>
>>> str(test_objects[0].property_)
'value'
>>> str(test_objects[1].property_)
'otherValue'
```

class quickbean.**AutoBean**

A decorator used to enhance the given class with an auto-generated equality, representation, cloning method and JSON encoder.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoBean
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
```

Which is strictly equivalent as :

```
>>> @quickbean.AutoEq
... @quickbean.AutoRepr
... @quickbean.AutoClone
... @quickbean.AutoToDict
... @quickbean.AutoToJson
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
```

class quickbean.**AutoEq**

A decorator used to enhance the given class with an auto-generated equality.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoEq
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
```

You will get an auto-generated equality taking all the properties available from your class :

```
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
```

```
... )
>>>
>>> test_object == TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='differentHiddenValue',
... )
True
>>>
>>> test_object == TestObject(
...     property_='value',
...     other_property='differentOtherValue',
...     _hidden_property='differentHiddenValue',
... )
False
```

An interesting thing to note here is that hidden properties -i.e. properties that begin with an underscore- are not taken into account.

It is also possible to exclude arbitrary properties with the 'exclude_properties' filter :

```
>>> @quickbean.AutoEq(quickbean.exclude_properties('excluded_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, excluded_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.excluded_property = excluded_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     excluded_property='excludedValue',
... )
>>>
>>> test_object == TestObject(
...     property_='value',
...     other_property='otherValue',
...     excluded_property='differentExcludedValue',
... )
True
```

If you prefer, it is even possible to do the opposite, that is to say, specifying the only properties to include with the 'only_include_properties' filter :

```
>>> @quickbean.AutoEq(quickbean.only_include_properties('property_', 'other_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, non_included_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.non_included_property = non_included_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     non_included_property='nonIncludedValue',
... )
>>>
>>> test_object == TestObject(
...     property_='value',
```

```
...     other_property='otherValue',
...     non_included_property='differentNonIncludedValue',
... )
True
```

class quickbean.**AutoRepr**

A decorator used to enhance the given class with an auto-generated representation.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoRepr
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
```

You will get an auto-generated representation taking all the properties available from your class :

```
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> repr(test_object)
"TestObject(other_property='otherValue', property_='value')"
```

An interesting thing to note here is that hidden properties -i.e. properties that begin with an underscore- are not taken into account.

It is also possible to exclude arbitrary properties with the 'exclude_properties' filter :

```
>>> @quickbean.AutoRepr(quickbean.exclude_properties('excluded_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, excluded_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.excluded_property = excluded_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     excluded_property='excludedValue',
... )
>>>
>>> repr(test_object)
"TestObject(other_property='otherValue', property_='value')"
```

If you prefer, it is even possible to do the opposite, that is to say, specifying the only properties to include with the 'only_include_properties' filter :

```
>>> @quickbean.AutoRepr(quickbean.only_include_properties('property_', 'other_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, non_included_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.non_included_property = non_included_property
```

```
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     non_included_property='nonIncludedValue',
... )
>>>
>>> repr(test_object)
"TestObject(other_property='otherValue', property_='value')"
```

class quickbean.**AutoClone**

A decorator used to enhance the given class with an auto-generated cloning method.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoClone
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property=None):
...         self.property_ = property_
...         self.other_property = other_property
...
...         if _hidden_property:
...             self._hidden_property = _hidden_property
```

You will get an auto-generated cloning method taking all the properties available from your class :

```
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> test_object_clone = test_object.clone()
>>>
>>> sorted(test_object_clone.__dict__.items())
[('other_property', 'otherValue'), ('property_', 'value')]
```

An interesting thing to note here is that hidden properties -i.e. properties that begin with an underscore- are not taken into account.

It is also possible to exclude arbitrary properties with the 'exclude_properties' filter :

```
>>> @quickbean.AutoClone(quickbean.exclude_properties('excluded_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, excluded_property=None):
...         self.property_ = property_
...         self.other_property = other_property
...
...         if excluded_property:
...             self.excluded_property = excluded_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     excluded_property='excludedValue',
... )
>>>
>>> test_object_clone = test_object.clone()
```

```
>>>
>>> sorted(test_object_clone.__dict__.items())
[('other_property', 'otherValue'), ('property_', 'value')]
```

If you prefer, it is even possible to do the opposite, that is to say, specifying the only properties to include with the 'only_include_properties' filter :

```
>>> @quickbean.AutoClone(quickbean.only_include_properties('property_', 'other_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, non_included_property=None):
...         self.property_ = property_
...         self.other_property = other_property
...
...         if non_included_property:
...             self.non_included_property = non_included_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     non_included_property='nonIncludedValue',
... )
>>>
>>> test_object_clone = test_object.clone()
>>>
>>> sorted(test_object_clone.__dict__.items())
[('other_property', 'otherValue'), ('property_', 'value')]
```

Note that it is also possible to override some of the properties on fly :

```
>>> test_object_clone = test_object.clone(other_property='overriddenOtherValue')
>>>
>>> sorted(test_object_clone.__dict__.items())
[('other_property', 'overriddenOtherValue'), ('property_', 'value')]
```

class quickbean.**AutoToDict**

A decorator used to enhance the given class with an auto-generated object to dict converter.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoToDict
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
```

You will get an auto-generated object to dict converter taking all the properties available from your class :

```
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> sorted(test_object.to_dict().items())
[('other_property', 'otherValue'), ('property_', 'value')]
```

An interesting thing to note here is that hidden properties -i.e. properties that begin with an underscore- are not taken into account.

It is also possible to exclude arbitrary properties with the 'exclude_properties' filter :

```
>>> @quickbean.AutoToDict(quickbean.exclude_properties('excluded_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, excluded_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.excluded_property = excluded_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     excluded_property='excludedValue',
... )
>>>
>>> sorted(test_object.to_dict().items())
[('other_property', 'otherValue'), ('property_', 'value')]
```

If you prefer, it is even possible to do the opposite, that is to say, specifying the only properties to include with the 'only_include_properties' filter :

```
>>> @quickbean.AutoToDict(quickbean.only_include_properties('property_', 'other_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, non_included_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.non_included_property = non_included_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     non_included_property='nonIncludedValue',
... )
>>>
>>> sorted(test_object.to_dict().items())
[('other_property', 'otherValue'), ('property_', 'value')]
```

class quickbean.**AutoToJson**

A decorator used to enhance the given class with an auto-generated JSON encoder.

To use this decorator, you just have to place it in front of your class :

```
>>> import quickbean
>>>
>>> @quickbean.AutoToJson
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
```

You will get an auto-generated JSON encoder taking all the properties available from your class :

```
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
```

```
... )
>>>
>>> test_object.to_json_str()
'{"other_property": "otherValue", "property_": "value"}'
```

Many frameworks handle JSON with dictionaries, so you never have to parse JSON strings directly. In that case, you should use the 'to_json_dict' method instead :

```
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> list(sorted(test_object.to_json_dict().items()))
[('other_property', 'otherValue'), ('property_', 'value')]
```

An interesting thing to note here is that hidden properties -i.e. properties that begin with an underscore- are not taken into account.

It is also possible to exclude arbitrary properties with the 'exclude_properties' filter :

```
>>> @quickbean.AutoToJson(quickbean.exclude_properties('excluded_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, excluded_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.excluded_property = excluded_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     excluded_property='excludedValue',
... )
>>>
>>> test_object.to_json_str()
'{"other_property": "otherValue", "property_": "value"}'
```

If you prefer, it is even possible to do the opposite, that is to say, specifying the only properties to include with the 'only_include_properties' filter :

```
>>> @quickbean.AutoToJson(quickbean.only_include_properties('property_', 'other_property'))
... class TestObject(object):
...     def __init__(self, property_, other_property, non_included_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self.non_included_property = non_included_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     non_included_property='nonIncludedValue',
... )
>>>
>>> test_object.to_json_str()
'{"other_property": "otherValue", "property_": "value"}'
```

This decorator relies on the standard JSON encoder (https://docs.python.org/2/library/json.html). Values are then encoded according to this JSON encoder. But sometimes, it may be useful to customize how to encode

some particular properties. This is done through methods named as the corresponding properties suffixed with '_to_json_str' :

```
>>> @quickbean.AutoToJson
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
...
...     def other_property_to_json_str(self):
...         return json.dumps('%sToJson' % self.other_property)
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> test_object.to_json_str()
'{"other_property": "otherValueToJson", "property_": "value"}'
```

If you prefer handle JSON with dictionaries, use the '_to_json_dict' suffixed method instead :

```
>>> @quickbean.AutoToJson
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
...
...     def other_property_to_json_dict(self):
...         return '%sToJson' % self.other_property
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> test_object.to_json_str()
'{"other_property": "otherValueToJson", "property_": "value"}'
```

This solution is quite simple, but it has a little drawback. Since it directly relies on the object's properties, the encoding code cannot be reused somewhere else. If you want your encoding code to be reusable, use types instead. Types are entities set to fields named as the corresponding properties suffixed with '_json_type' that encode the taken property value through the available 'to_json' method :

```
>>> class CustomJsonType(object):
...     # noinspection PyMethodMayBeStatic
...     def to_json_str(self, value):
...         return json.dumps('%sToJson' % value)
>>>
>>> @quickbean.AutoToJson
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
```

```
...
...     other_property_json_type = CustomJsonType()
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> test_object.to_json_str()
'{"other_property": "otherValueToJson", "property_": "value"}'
```

If you prefer handle JSON with dictionaries, use the 'to_json_dict' method instead :

```
>>> class CustomJsonType(object):
...     # noinspection PyMethodMayBeStatic
...     def to_json_dict(self, value):
...         return '%sToJson' % value
>>>
>>> @quickbean.AutoToJson
... class TestObject(object):
...     def __init__(self, property_, other_property, _hidden_property):
...         self.property_ = property_
...         self.other_property = other_property
...         self._hidden_property = _hidden_property
...
...     other_property_json_type = CustomJsonType()
>>>
>>> test_object = TestObject(
...     property_='value',
...     other_property='otherValue',
...     _hidden_property='hiddenValue',
... )
>>>
>>> test_object.to_json_str()
'{"other_property": "otherValueToJson", "property_": "value"}'
```

Sometimes, it may be very useful to directly encode a list of objects instead of having to encode them one by one. This is done through the 'list_to_json_str' method :

```
>>> @quickbean.AutoToJson
... class TestObject(object):
...     def __init__(self, property_):
...         self.property_ = property_
>>>
>>> test_object = TestObject(
...     property_='value',
... )
>>> other_test_object = TestObject(
...     property_='otherValue',
... )
>>>
>>> TestObject.list_to_json_str(test_object, other_test_object)
'[{"property_": "value"}, {"property_": "otherValue"}]'
```

A